Performance Analysis of Fermat Factorization Algorithms

Hazem M. Bahig[1]*, Mohammed A. Mahdi[2], Khaled A. Alutaibi[3], Amer AlGhadhban[4], Hatem M. Bahig[5]
Computer Science and Information Department, College of Computer Science and Engineering, University of Ha'il, Ha'il, KSA[1,2]
Computer Science Division, Mathematics Department, Faculty of Science, Ain Shams University, Cairo, Egypt[1,5]
Computer Engineering Department, College of Computer Science and Engineering, University of Ha'il, Ha'il, KSA[3]
Electrical Engineering Department, College of Engineering, University of Ha'il, Ha'il, KSA[5]

***Abstract*—The Rivest-Shamir-Adleman (RSA) cryptosystem is one of the strong encryption approaches currently being used for secure data transmission over an insecure channel. The difficulty encountered in breaking RSA derives from the difficulty in finding a polynomial time for integer factorization. In integer factorization for RSA, given an odd composite number *n*, the goal is to find two prime numbers *p* and *q* such that *n* = *p q*. In this paper, we study several integer factorization algorithms that are based on Fermat's strategy, and do the following: First, we classify these algorithms into three groups: Fermat, Fermat with sieving, and Fermat without perfect square. Second, we conduct extensive experimental studies on nine different integer factorization algorithms and measure the performance of each algorithm based on two parameters: the number of bits for the odd composite number *n*, and the number of bits for the difference between two prime factors, *p* and *q*. The results obtained by the algorithms when applied to five different data sets for each factor reveal that the algorithm that showed the best performance is the algorithms based on (1) the sieving of odd and even numbers strategy, and (2) Euler's theorem with percentage of improvement of 44% and 36%, respectively compared to the original Fermat factorization algorithm. Finally, the future directions of research and development are presented.**

***Keywords*—Integer factorization; Fermat's algorithm; RSA; factorization with sieving; perfect square**

I. Introduction

The Rivest-Shamir-Adleman (RSA) cryptosystem is one of the most famous and secure cryptosystems currently available. It was designed to encrypt plain text into cipher text in as strong a manner as possible. The RSA system is a type of public-key cryptosystem that is based on two different keys: a public key that is used for encryption and a private key that is used for decryption.

The main steps in the RSA cryptosystem are as follows [1,2]:

1) Generate two random distinct prime numbers of large and equal size, p and q, and then construct an odd composite number $n = p\,q$.

2) Calculate the Euler function $\Phi(n) = (p-1)(q-1)$.

3) For the encryption procedure, choose the exponent number e that is greater than 1 and less than $\Phi(n)$ such that $\gcd(e, \Phi(n)) = 1$. Then apply the modular exponentiation formula on the message m to generate a secret message c as follows:

$$c = m^e \bmod n$$

4) For the decryption procedure, find the integer d that is greater than 1 and less than $\Phi(n)$ such that $e\,d \bmod \Phi(n) = 1$. Then apply the modular exponentiation formula on the secret message c to generate a message m as follows:

$$m = c^d \bmod n$$

The RSA cryptosystem includes two mathematical operations that are opposite to each other. The first operation is multiplication, which is easy to compute. The running time to compute the product of two numbers is $O(b^2)$ in the worst case, where b is the size of each number. This type of operation is important for computing the modular exponentiation [3, 4] to reduce the computation time of the exponentiation. The second operation is a process that involves finding two prime factors p and q from an odd composite number n. This process is called integer factorization [5]. If we can factor n to p and q, then we can compute $\Phi(n)$ and then d. Consequently, the encrypted message c can be decrypted. Hence, the integer factorization problem is important in cryptography. Therefore, solving this problem in an efficient timeframe leads to breaking the RSA. In other side, the difficulty in finding a polynomial time for the factorization leads to difficulty in breaking the RSA cryptosystem [6, 7, 8, 9].

Moreover, the integer factorization problem is important from the point of view of complexity theory. Until now, the integer factorization problem has not been considered to belong to the class of P problems. Also, there is no proof that the integer factorization problem belongs or does not belong to the class of NP-complete problems. From a review of the literature, it seems that the best time complexity for factoring an odd composite number is

$$exp\left(\sqrt[3]{64/9} + o(1)(\ln n)^{1/3} (\ln \ln n)^{2/3}\right)$$

using the general number field sieve (GNFS) algorithm [1, 10].

A large number of algorithms have been proposed in order to attempt to factor an odd composite number. These algorithms can be categorized as either general or special-purpose algorithms. The general-purpose group contains

*Corresponding Author

integer factorization algorithms that have a running time that depends on the size of an odd composite number only. This group includes integer factorization algorithms that are based on various strategies, such as continued fraction factorization, Shanks's square forms factorization, Dixon's algorithm, the quadratic sieve algorithm and the GNFS algorithm [1, 2, 10].

On the other hand, the special-purpose group contains integer factorization algorithms that have a running time that depends on the size of an odd composite number and its properties. For example, the trial division method is an efficient algorithm for factorization when an odd composite contains a small prime factor. Besides trial division, the special-purpose group contains various other techniques, such as Fermat factorization, wheel factorization, Pollard's p-1, Euler factorization, and the Lenstra elliptic curve [1, 2, 10, 11].

In this paper, we are interested in the special-purpose group because the aim is to study the performance of algorithms that are based on Fermat's factorization concept. Fermat proposed a factorization algorithm that is based on representing the odd composite number as the difference between two squares. The main advantage of Fermat's factorization technique is that it is able to factor an odd composite number, n, in a very fast time, i.e., almost instantaneously, when the difference between two factors is $\Delta = \sqrt[4]{n}$ [6, 12]. This means that if the size of n is b bits and the difference between two factors is $\Delta = \sqrt[4]{n}$, then the following are true: (1) The two prime factors have the same size, i.e., each prime factor has size $b/2$; (2) The number of common bits between the two prime factors is $b/4$, and these bits should be the most significant bits. These two conditions are known as the domain of the efficiency of Fermat's algorithm, (DEF).

Many algorithms have been proposed that are based on Fermat's factorization concept [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]. The goal of these algorithms is to improve the running time of the original Fermat algorithm in finding prime factors. Two categories of factor have an effect on the efficiency of these algorithms. The first category is related to the properties of the input, which includes the size of the odd composite number, b, and the difference between two factors, Δ. The second category is related to the natural of the algorithm itself such as the search strategy it uses to find the solution and the number of high-cost operations included in the algorithm.

In general, the improved Fermat algorithms can be classified into two classes. The first class contains algorithms based on the concept of an estimated prime factor and uses different techniques such as continued fraction method [28] or considering n as a special form $6k \pm 1$, where k is any integer [23]. However, the techniques in this class cannot factor some odd composite numbers, so they cannot be considered as general methods for Fermat factorization. The second class contains algorithms [11, 14, 15, 17, 18, 19, 20, 21, 22, 24, 25, 26, 27, 29] that can be applied to any odd composite number and are based on (1) replacing the high-cost operation, i.e., the perfect square in Fermat's method, with a low-cost operation or on (2) reducing the space searched to find the solution. It should also be noted that there is another strategy [13, 30] that falls outside the scope of our research, which involves speeding up the running time of Fermat's algorithm that is based on a different platform such as high-performance computing [13, 33].

In this paper, we are interested in the integer factorization algorithms that are belong to the second class. From our analysis of these techniques, we made the following observations:

1) The experimental studies for most of these algorithms were implemented when the size of an odd composite was less than 64 bits [15, 20, 21, 26, 32], for example, n = 84449 × 21121 =1783647329. This number of bits is small compared to that required in cryptography. Also, the time consumed for an operation increases with increase the size of data, especially for high-cost operations.

2) Many of the experimental studies for some of these algorithms were implemented when the difference between the two factors did not belong to the DEF. This means that any comparison between Fermat's algorithm and these algorithms is not realistic because it has been proved that the Fermat factorization method is not efficient outside the DEF. For example,n=19710741 × 531349691=1047329636821139813 [20].

3) The efficiency of some of these algorithms was measured based on a few data or on some examples, rather than on different values for b and Δ, see for example, [22, 25]. This means that there is no exhaustive study that compares two or more integer factorization algorithms that are based on Fermat factorization concept by applying them to different data distributions in the DEF.

4) A few steps in some of these algorithms required some optimization due to the cost of the operation to manipulate a large data size.

Consequently, we are interested in undertaking an experimental study on most of the integer factorization algorithms that are based on Fermat's concept in order to answer the following:

Q1) Which one of the Fermat factorization algorithms is efficient for a large size of n and a small value of Δ?

Q2) What is the effect of increasing the value of Δ with a fixed size of n for each of the studied algorithms?

Q3) Many integer factorization algorithms have the same number of iterations, theoretically, but which one is the fastest over different data distributions?

To the best of our knowledge, there is no sufficient comparative study for Fermat factorization algorithms, especially with regard to the effect of the use of factors b and Δ on performance. Also, our study compares the performance of nine different integer factorization algorithms in order to determine which has the fastest running time.

The remainder of the paper is structured as follows: In Section II, we provide the methodology used to verify the objectives of this study. In Section III, we provide an overview of the different integer factorization algorithms that are based on Fermat factorization, including the pseudocode of each

algorithm. Additionally, we classify these algorithms based on the techniques used into three groups. In Section IV, we describe the experimental study undertaken to compare and measure the performance of different integer factorization algorithms. Also, we present an analysis of the results produced by the experimental study. Finally, in Section V, we draw some conclusions from this work and highlight open questions that remain to be answered in future studies.

II. Methodology

To measure the performance of the integer factorization algorithms that are based on Fermat's strategy, we followed a methodology that consisted of five stages:

1) Determine the different strategies that need to be used to factor an odd composite integer into two prime factors according to Fermat's concept;

2) Determine the language and packages to use to verify the goal of the study;

3) Optimize the implementation of the selected integer factorization algorithms based on the platform used in the experimental studies;

4) Generate a dataset to use to measure the performance of the selected algorithms, especially when applied to large data sizes; and

5) Measure and analyze the performance of the selected algorithms.

Here we discuss, briefly, each of the above stages. In the first stage, we studied the different algorithms that use Fermat's strategy to find the two prime factors for an odd composite number. We classified these algorithms into groups based on the concept used in the algorithms. For each algorithm, we identified the main idea, the pseudocode and the expected number of iterations to find the solution. The details of this stage are covered in Section III.

In the second stage, we selected the language and package to use in our study, namely, C++ language and the GNU Multiple Precision (GMP) arithmetic library [34]. We decided to use C++ language because the performance of this language is fast compared to other languages such as Java. In other side, to execute any operation in the RSA system requires a number of size 1024 bits. However, the size of the integer type in C++ language does not support this objective because it is limited to 64 bits. Therefore, we decided to use the GMP library because it is designed to support applications such as cryptography and computational algebra that involve large-sized numbers. Furthermore, the library has the following advantages: (1) It contains a large number of functions to manipulate integers and other types; (2) the functions in the library are fast compared to those of other tools; and (3) there is no limitation to the size of number, so we can manipulate a number that is greater than 1024 bits in size.

It should be noted that, in our implementation, we used only the data type mpz_t that is used to manipulate GMP integers. The library contains a group of functions to manipulate GMP integers, such as (1) initializing and assigning GMP integers, (2) integer arithmetic and division, (3) integer roots, and (4) integer comparisons.

In the third stage, we focused on optimizing each algorithm, if required, in order to run the algorithm in a fast way. The reasons for doing this were as follows: (1) The cost of the operation for a large integer size is significantly different than that for small integers of less than 64 bits for the same operation [35, 36]; and (2) for some algorithms, we needed to rewrite a few of the statements to increase the performance of an algorithm. The details of this stage are provided in Section III.

In the fourth stage, we employed a method to generate an odd composite number consisting of a product of two prime numbers. In this method, the generation of the two prime factors is based on the two factors of the DEF [6, 12]. The first factor is the size of the odd composite number n. Suppose that the size of n is b, where b is the number of bits. Therefore, when we generate an odd composite number n of size b bits, we first generate two prime numbers p and q, each of size $b/2$, and then we multiply both of them, i.e., $n = p\ q$. The second factor is the difference between the prime factors p and q, Δ. This factor is important because the running time of Fermat's strategy increases with an increase in the value of Δ.

The generation of one data (an odd composite number), GD, consists of the following steps: The first step is to determine the number of bits for n, b, and the number of bits for the difference between two factors, Δ. The second step is to generate a prime number of size $b/2$, say p. The third step is to generate a random number of size Δ and then add it to the first prime number, say x. The fourth and final step involves generating the second prime number, say q, greater than x such that the size of the difference between p and q is Δ.

For more accuracy in measuring the performance of the different algorithms, we repeated the previous steps for GD by adopting the following procedure: First, we fixed the value of b and Δ. Second, we applied the steps for generating two prime factors as in GD, i.e., from the second step to the fourth step in GD, t times, where t represents the number of different instances that have the same value of b and Δ. Third, we repeated the execution of the first and second steps of the procedure with increasing values of Δ in increments of 5 bits until $\Delta + 20$. The reason for setting a maximum value of Δ is that the running time of all the integer factorization algorithms based on Fermat's strategy increases rapidly with an increase in Δ. Fourth, we increased the value of b and then repeated all the previous steps.

The last important task in the data generation stage was to verify that the generated data were correct as follows:

1) Each prime factor, p, should be verified such that $2^{(b/2)-1} < p < 2^{(b/2)}$.

2) The difference between the two factors should be verified such that $2^{\Delta-1} \leq |p-q| < 2^{\Delta}$.

3) The prime factors used for the fixed values of b and Δ should be as different as possible.

The fifth and final step in the methodology involved measuring the performance of the integer factorization algorithms that are based on Fermat's strategy. The performance of these algorithms was mainly measured by

computing the execution time. Hence, for fixed values of b and Δ, the running time for the algorithm A, $T_A(b,\Delta)$, computed using the following formula:

$$T_A(b,\Delta) = \frac{1}{t}\sum_{i=1}^{t} T_A(b,\Delta)_i$$

where $T_A(b,\Delta)_i$ is the running time of the algorithm A for the instance number i using input data b and Δ. Note that two instances, $(b,\Delta)_i$ and $(b,\Delta)_j$, are different if the odd composite number, n_i, of size b for the instance i is different than the odd composite number, n_j, of size b for the instance j. Additionally, with respect to the issue of memory consumption, all the algorithms required a constant number of auxiliary variables, so there was no need to measure this factor experimentally.

III. Classification of Fermat Algorithms

In this section, we provide an overview of the different algorithms for integer factorization that are based on Fermat's concept. For each algorithm, we discuss the main idea and steps of the algorithm, and then we give the pseudocode of the algorithm.

Without loss of generality, for all algorithms, we assume that the integer number n is odd and is a product of two prime numbers p and q, where $p > q$. The main idea of Fermat's algorithm is that the integer number n can be expressed as a difference between two square numbers, x and y. Formally, the odd integer number n can be written as follows:

$$n = x^2 - y^2 \qquad (1)$$

Also, the relation between the two prime factors and the two square numbers is as follows:

$$p = x + y \text{ and } q = x - y \qquad (2)$$

Different strategies have been proposed to factorize an odd composite number into two prime numbers based on Fermat's concept. All the algorithms start the search with an initial value of x and try to find the value of y such that $y^2 = x^2 - n$. So, the main issues in Fermat's strategy are (1) how to reduce the search space of x and (2) how to reduce the cost of the perfect squaring operation, where a simple test for a perfect square for x includes two operations: calculating the square root for x, say r, and testing whether the value of r is an integer or not.

We can classify the algorithms of integer factorization that are based of Fermat's strategy into three main groups. The first group employs a direct approach which starts from the minimum value of x and uses a perfect square operation. We named this group the Fermat factorization group because it is based on directly applying the concept proposed by Fermat. The second group is based on sieving or pruning some elements in the search space, so the algorithm does not apply the squaring operation or/and perfect squaring operation on those elements. We named this group the Fermat with sieving group. The third group is based on removing the main operation of the Fermat factorization algorithm which is the perfect squaring operation, so we named this group the Fermat without perfect squaring group. The main ideas and steps of these three groups of algorithms are outlined in the following subsections.

A. *The Fermat Factorization Group*

The Fermat factorization group contains many algorithms which are based mainly on the perfect squaring operation [14, 15, 17]. The first and main algorithm in this group is based on rewriting Eq. (1) as $y^2 = x^2 - n$ and starts by assuming that the value of x is $\lfloor\sqrt{n}\rfloor + 1$. Then the algorithm tests whether the value of $x^2 - n$ is a perfect square. If the value of y is an integer and equal to $\sqrt{x^2 - n}$, the search is terminated. When $x^2 - n$ is not a perfect square, the algorithm increases the value of x by 1 and follows the same procedure until it finds the value of y.

All the steps of the algorithm are shown in Algorithm FF. The algorithm is very straightforward and contains simple operations, except for the perfect square operation. The running time of the algorithm is based on two factors. The first factor is the cost of the perfect square operation and the second is the number of iterations for the While-loop which is equal to $x - (\lfloor\sqrt{n}\rfloor + 1)$ in the worst case, where x satisfies that the term $x^2 - n$ is a perfect square and equals $(p+q)/2$.

Algorithm FF (Fermat's Factorization)
Input: n is a positive odd number.
Output: p and q are two prime numbers such that $n = p\,q$.
Begin
1. $x = \lfloor\sqrt{n}\rfloor + 1$
2. $y = x^2 - n$
3. While (y is not a perfect square) do
4. $\quad x = x + 1$
5. $\quad y = x^2 - n$
6. End while
7. $p = x + \sqrt{y}$
8. $q = x - \sqrt{y}$

End.

Remarks:

1) Many modified algorithms [14, 15, 17] have been proposed to improve the FF algorithm while retaining the two operations, squaring and perfect squaring. The modifications are based on rewriting the Fermat factorization formula and then searching for the solution. For example, in [17], the formula is rewritten as $z^2 = (x + \lceil\sqrt{n}\rceil y)^2 - ny^2$, with a small x and y, and the goal is to find the solution (x, y, z). In [15], the algorithm is modified in order to achieve the goal of finding a solution for $z^2 = (\lceil\sqrt{n\,i}\rceil)^2 - n\,i$, where i starts with a value of 1. The modified algorithm, FF1, contains four operations before testing the perfect square. These operations are: multiplying n with i, squaring the root of (n i), say s, squaring s, and calculating the modulus of n. This means that, in general, the modified algorithm contains more operations than the FF algorithm. Therefore, we neglected these modifications in our general comparison of the different types of integer factorization algorithms (see Section IV).

2) In order to optimize the code of the FF algorithm we decided to do the following: (*i*) To compute x^2, we multiplied x with itself to get better performance instead of using the predefined function power for the exponent 2; (*ii*) in the case of the perfect square operation, we used the predefined function in GMP because we considered that this would be better than computing the square root of the number and then testing whether the results are integers or not.

Algorithm FF1 (Modified Fermat's Factorization)
Input: *n* is a positive odd number.
Output: p and q are two prime numbers such that $n = p\,q$.
Begin
1. $i = 1$
2. *found* =false
3. While ($found \neq$ true) do
4. $s = \lceil \sqrt{n\,i} \rceil$
5. $m = s^2 \bmod n$
6. If IsSquare(*m*) then
7. $t = \sqrt{m}$
8. *found* =true
9. return gcd(*n*,*s*-*t*)
10. End if
11. $i = i + 1$
12. End while
End.

B. The Fermat with Sieving Group

The sieving strategy is a method that is used to remove the impossible solutions so that the algorithm does not consider them during the search process. The algorithms that apply this strategy for Fermat factorization can be classified into two classes of techniques.

The first class of techniques ignores the perfect squaring operation in some cases. This means that before testing whether integer *y* is a perfect square or not, the technique tests whether *y* satisfies a certain condition. If integer *y* meets this condition, the technique does not test whether *y* is a perfect square and goes instead to the next value of *x*. Otherwise, the technique tests whether *y* is a perfect square or not.

The second class of techniques ignores the squaring operation and consequently the perfect square operation. This means that before calculating x^2, the strategy tests whether *x* satisfies a certain condition. If integer *x* meets this condition, the technique ignores all the subsequent steps, i.e., squaring, subtraction, and perfect squaring, and goes to the next value of *x*. Otherwise, the technique tests whether *y* is a perfect square or not.

1) Class 1: Ignoring the perfect square

a) Sieving with modulus: One of the techniques used in sieving is the modulus operation, mod. The idea behind using the modulus arithmetic operation is to exclude all integers, x, that are definitely not perfect squares before applying the perfect squaring operation [22, 24].

TABLE I. VALUES OF X AND X^2

x	x^2	x	x^2
$x_{l-1}x_{l-1}\dots x_1 0$	$x'_{2l-1}x'_{l-1}\dots x'_1\mathbf{0}$	$x_{l-1}x_{l-1}\dots x_1 5$	$x'_{2l-1}x'_{l-1}\dots x'_1\mathbf{5}$
$x_{l-1}x_{l-1}\dots x_1 1$	$x'_{2l-1}x'_{l-1}\dots x'_1\mathbf{1}$	$x_{l-1}x_{l-1}\dots x_1 6$	$x'_{2l-1}x'_{l-1}\dots x'_1\mathbf{6}$
$x_{l-1}x_{l-1}\dots x_1 2$	$x'_{2l-1}x'_{l-1}\dots x'_1\mathbf{4}$	$x_{l-1}x_{l-1}\dots x_1 7$	$x'_{2l-1}x'_{l-1}\dots x'_1\mathbf{9}$
$x_{l-1}x_{l-1}\dots x_1 3$	$x'_{2l-1}x'_{l-1}\dots x'_1\mathbf{9}$	$x_{l-1}x_{l-1}\dots x_1 8$	$x'_{2l-1}x'_{l-1}\dots x'_1\mathbf{4}$
$x_{l-1}x_{l-1}\dots x_1 4$	$x'_{2l-1}x'_{l-1}\dots x'_1\mathbf{6}$	$x_{l-1}x_{l-1}\dots x_1 9$	$x'_{2l-1}x'_{l-1}\dots x'_1\mathbf{1}$

We can apply this technique as follows: For any integer *x*, we can represent *x* in decimal form as $x_{l-1}x_{l-1}\dots x_1x_0$, where *l* represents the number of decimal digits in *x*. It is clear that the last, right-most, digit of *x* is either 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. The last digit for squaring *x* can be calculated by taking the modulus of 10 and is equal to 0, 1, 4, 5, 6, or 9, see the bold digit in Table I. On the other hand, no squaring number has a last digit of 2, 3, 7 or 8. Therefore, we compute $r = y \bmod 10$ and if the value of r is 2, 3, 7, or 8, then there is no need to test whether y is a perfect square or not, and so we can go to the next value of *x* directly.

The complete steps of the algorithm is shown in Algorithm FM10 [22, 24]. The algorithm is similar to the FF algorithm but contains two extra statements. The first statement computes the modulus of 10 for the term $x^2 - n$, and the second tests the result of using the modulus, see line 7. The running time of the algorithm is similar to that of the FF algorithm, except (1) two extra statements, see lines 6 and 7, and (2) the algorithm uses fewer perfect square operations based on the truth value of the condition.

Algorithm FM10 (Fermat Sieving by Modulus 10)
Input: *n* is a positive odd number.
Output: p and q are two prime such that $n = p\,q$.
Begin
1. $x = \lfloor \sqrt{n} \rfloor$
2. *found*=false
3. While (Not *found*) do
4. $x = x + 1$
5. $y = x^2 - n$
6. $r = y \bmod 10$
7. If Not ($r = 2$ or $r = 3$ or $r = 7$ or $r = 8$) then
8. If (y is a perfect square) then
9. *found*=True
10. End if
11. End if
12. End while
13. $p = x + \sqrt{y}$
14. $q = x - \sqrt{y}$
End.

Remarks:

1) The statement in line 7 can be rewritten as follows:

If $(r = 0$ or $r = 1$ or $r = 4$ or $r = 5$ or $r = 6$ or $r = 9)$ then

However, this statement contains six comparisons at most, whereas the statement in line 7 contains four comparisons at most, but the running time for two versions is almost similar.

2) We can use a modulus of 15, 20 or 30 instead of 10. To study the effect of changing the value of the modulus on the performance of the algorithm, we changed the modulus of 10 to a modulus of 20 and named this method FM20. Using this approach, the accepted cases to ignore the test for the perfect square occur when the remainder of (r=y mod 20) are 0, 1, 4, 5, 9, or 16. The steps of the FM20 algorithm are similar to those of FM10, except that line 7 is replaced with:

If Not $(r = 0$ or $r = 1$ or $r = 4$ or $r = 5$ or $r = 9$ or $r = 16)$ then

We studied the effect of this change experimentally, see Section IV.

b) Sieving with odd & even: In the FF algorithm, the values of x are $\lfloor\sqrt{n}\rfloor + 1$, $\lfloor\sqrt{n}\rfloor + 2$, $\lfloor\sqrt{n}\rfloor + 3$, $\lfloor\sqrt{n}\rfloor + 4$,…. This means that the values of x are odd and even numbers. Another sieving technique that can be applied in integer factorization algorithms is based on ignoring all the even (or all the odd) numbers of x if the integer n satisfies a certain condition. The idea behind using the even and odd property is based on the following rules [19, 29]:

1) Any odd integer n can be expressed as $n = 4k \pm 1, n \geq 3$.

2) For $n = 4k \pm 1$, we have two cases: (*i*) when $n = 4k + 1$, then x is odd and y is even, and (*ii*) when $n = 4k - 1$, then x is even and y is odd.

The algorithm consists of four main steps. The first step determines the form of n as either $4k + 1$ or $4k - 1$. The second step determines the type (even or odd) of x and y. The third step determines the start value of x in the case of whether x is odd or even. The fourth step applies the steps of FF algorithm with updating the value of x with 2.

The complete steps of the algorithm are shown in Algorithm FOE. To determine the formula of n, the algorithm computes the remainder of dividing n with 4 and then tests if the remainder is equal to 1 or not, see lines 2–3. If the remainder is equal to 1, this means that $n = 4k + 1$, otherwise, $n = 4k - 1$. Lines 4–5 and 8–9 determine the type, odd or even, and the start value of the integer x. The remainder of the algorithm is similar to the FF algorithm except when the value of $x^2 - n$ is not a perfect square, the algorithm updates the value of x by 2 instead of 1 because when x is even (or odd), the next even (or odd) number of x can be found by adding 2 to x. Hence, the number of iterations of the FOE algorithm is half that of the FF algorithm because the FOE algorithm updates the value of x by 2, whereas the FF algorithm updates the value of x by 1.

Algorithm FOE (Fermat Sieving using Odd & Even)
Input: n is a positive odd number.
Output: p and q are two prime numbers such that $n = p\,q$.

```
 Begin
  1.   x = ⌊√n⌋ + 1
  2.   r = n mod 4
  3.   If (r = 1) then
  4.        If (x is even) then
  5.             x = x + 1
  6.        End if
  7.   Else
  8.        If (x is odd) then
  9.             x = x + 1
 10.        End if
 11.   End if
 12.   y = x² − n
 13.   While (y is not a perfect square) do
 14.        x = x + 2
 15.        y = x² − n
 16.   End while
 17.   p = x + √y
 18.   q = x − √y
 End.
```

---

3) Class 2: Ignoring the Squaring: Another important improvement that has been made to Fermat's algorithm is the ignoring of some elements in the search space before squaring the value of x. Two algorithms have been proposed to achieve this goal. The two algorithms are based on analyzing the relation between the value of (x mod 10) and the value of (n mod m), where m may be 10, 15, 20, or 30.

For any integer x, the value of (x mod m) is either 0, 1, 2, …, m-2, or m-1. When n is odd and not divisible by 5 and m = 10, the value of (n mod 10) is 1, 3, 7, or 9, and when m = 20, the value of (n mod 20) is 1, 3, 7, 9, 11, 13, 15, 17, or 19. We ignored the value 5, because if (n mod 20)=5, then 5 is a factor of n.

TABLE II. DIFFERENT CASES FOR (X MOD 10) AND (N MOD 10)

x mod 10	**Results of $(x^2 - n)$ mod 10 when n mod 10 equal**			
	1	**3**	**7**	**9**
0	9	7	3	1
1	0	8	4	2
2	3	1	7	5
3	8	6	2	0
4	5	3	9	7
5	4	2	8	6
6	5	3	9	7
7	8	6	2	0
8	3	1	7	5
9	0	8	4	2

Based on the two values, (x mod 10) and (n mod m), we can determine when there is no need to compute x^2. Table II displays the different cases for the relation between (x mod 10) and (n mod 10), see the gray-colored cells. Therefore, the accepted cases to apply the idea of ignoring the squaring operation are as follows [25]: (1) n mod 10=1 and x mod 10=2, 3,7 or 8. (2) n mod 10=3 and x mod 10=0, 1, 4, 5, 6 or 9. (3) n mod 10=7 and x mod 10=0, 2, 3, 5, 7 or 8. (4) n mod 10=9 and x mod 10=1, 4, 6 or 9.

The complete steps of this technique are shown in Algorithm FM1010 [24]. Initially, we determine the initial value of x and (n mod 10). Then the algorithm choose one of the four cases, as described previously. The number of iterations of the FM1010 algorithm is similar to that of the FF algorithm, but some of these iterations ignore the squaring and perfect squaring operations.

Algorithm FM1010 (Fermat Sieving by two Modulus 10)
Input: n is a positive odd number.
Output: p and q are two primes such that $n = p\,q$.

```
Begin
1.  x = ⌊√n⌋
2.  found=false
3.  x = x + 1
4.  r1 = n mod 10
5.  While (Not found) do
6.      r2 = x mod 10
7.      If (r1 = 1) then
8.          If Not (r2 = 2 or 3 or 7 or 8) then
9.              y = x^2 − n
10.             If (y is a perfect square) then
11.                 found=True
12.             Else
13.                 x = x + 1
14.             End if
15.         Else
16.             x = x + 1
17.         End if
18.     Else
19.       If (r1 = 3) then
20.         If Not (r2 = 0 or 1 or 4 or 5 or 6 or 9) then
21.                         Similar to lines 9-17
22.       Else
23.         If (r1 = 7) then
24.           If Not (r2 = 0 or 2 or 3 or 5 or 7 or 8) then
25.                         Similar to lines 9-17
26.         Else
27.           If (r1 = 9) then
28.             If Not (r2 = 1 or 4 or 6 or 9) then
29.                         Similar to lines 9-17
30.         End if
31. End while
32. p = x + √y
33. q = x − √y
End.
```

Similarly, we construct the accepted cases for applying the idea of sieving for the relation between (x mod 10) and (n mod 20) as follows [25]: (1) n mod 20=1 and x mod 10=1, 5 or 9. (2) n mod 20=3 and x mod 10=2 or 8. (3) n mod 20=7 and x mod 10=4 or 6. (4) n mod 20=9 and x mod 10=3, 5 or 7. (5) n mod 20=11 and x mod 10=0, 4 or 6. (6) n mod 20=13 and x mod 10=3 or 7. (7) n mod 20=17 and x mod 10=1 or 9. (8) n mod 20=19 and x mod 10=0, 2 or 8.

Using these cases, we constructed another sieving method named FM1020 algorithm. The pseudocode of the FM1020 algorithm is similar to that of FM1010, except that the While-loop contains eight If-statement, where each If-statement represents a case from the eight cases of the relation between (x mod 10) and (n mod 20) [25].

Remark: Another modified algorithm was proposed by Somsuk, and Tientanopajai [27] and based on analyzing the last m digits of the modulus, where $m \geq 2$. The main drawback of the modified algorithm is the number of different subroutines used is very large when m is large.

C. The Fermat without Perfect Squaring Group

The algorithms in this group do not use the perfect squaring operation during the search for the two prime factors. Two main techniques have been proposed for use in integer factorization algorithms that do not employ perfect squaring.

The first algorithm is based on finding two integers x and y such that the difference between two square numbers is $4n$, i.e., $4n = x^2 - y^2$. This formula can be rewritten as follows [27, 28]:

$$r = x^2 - (y^2 + 4n)$$

The algorithm starts the search with $x = 2\lceil\sqrt{n}\rceil$ and $y = 0$, and then computes the value of r. Based on the value of r, the algorithm executes one of the following cases [27, 28]:

1) Case 1: $r = 0$. This means that $4n$ is equal to the difference between two squares and the search process is terminated. Therefore, the two prime factors are $p = (x + y)/2$ and $q = (x - y)/2$.

2) Case 2: $r < 0$. This means that the value of the term x^2 is less than the term $(y^2 + 4n)$. Therefore, the algorithm increases the value of x by 2 because the value of the term $(p + q)$ is an even number. Also, the algorithm increases the value of r by $(4x + 4)$ because this value represents the difference between the squares of the next and the current value of x.

3) Case 3: $r > 0$. This means that the value of the term x^2 is greater than the term $(y^2 + 4)$. Therefore, the algorithm increases the value of y by 2 because the value of the term $(p - q)$ is an even number. Also, the algorithm decreases the value of r by $(4y + 4)$ because this value represents the difference between the squares of the next and current value of y.

The complete steps of the algorithm are shown in Algorithm FnPS. All the operations of the algorithm are simple and the running time of the algorithm is based only on the

number of iterations of While-loop that is dependent on two conditions: $r < 0$ and $r > 0$. The number of iterations for the first inner while-loop that is based on the condition $(r < 0)$ is $\frac{p+q}{2} - (\lfloor\sqrt{n}\rfloor + 1)$, while the number of iterations for the second inner while-loop that is based on the condition $(r > 0)$ is $\frac{p-q}{2}$, because the start value of *y* is 0. Therefore, the total number of iterations of the FnPS algorithm is $p - (\lfloor\sqrt{n}\rfloor + 1)$.

Algorithm FnPS (Fermat with no Perfect Squares)
Input: *n* is a positive odd number.
Output: p and q are two prime numbers such that $n = p\,q$.

```
Begin
  1.  x = 2⌈√n⌉
  2.  y = 0
  3.  While (r ≠ 0) do
  4.      While (r < 0) do
  5.          r = r + (4x + 4)
  6.          x = x + 2
  7.      End while
  8.      While (r > 0) do
  9.          r = r − (4y + 4)
  10.         y = y + 2
  11.     End while
  12. End while
  13. p = (x + y)/2
  14. q = (x − y)/2
End.
```

The second algorithm in this group is based on removing the perfect squaring operation by using modular multiplication. The method of modification is based on Euler's theorem which is given by the following formula [20]:

$$a^{\Phi(n)} \equiv 1 \bmod n$$

where *n* is a positive integer number, *a* is an integer such that *a* is relatively prime with *n*, and $\Phi(n)$ is the Euler's totient function that is equal to $(p-1) \times (q-1) = n - (p+q) + 1$.

The complete steps of the algorithm are shown in Algorithm FE [20]. The algorithm starts the computation by initializing $x = 2\lceil\sqrt{n}\rceil$ and selecting a positive integer *c* which is relatively prime with *n*. Based on the value of *c*, the algorithm computes the inverse, *a*, and the square, *s*, of *c* in modulo *n*. Then the algorithm applies Euler's theorem, where the value of $\Phi(n) = n - x + 1$. If the result, *t*, equals 1 then the values of *x* and *y* are determined and the solution is found. Otherwise, the algorithm sets a flag with false. In which case the algorithm repeats the following steps until Euler's theorem is verified, i.e., when *t* = 1. The first step in the repetition is to compute a new value *t* by multiplying it with *s* and taking a module if it is required. The second step is to update the value of *x* by increasing it by a value of 2. Finally, the value of the two prime factors are $x + y$ and $x - y$. The total number of iterations of this algorithm is similar to that of the FF algorithm, but the perfect squaring operation and only one square root operation is required.

Algorithm FE (Fermat-Euler)
Input: *n* is a positive odd number.
Output: p and q are two prime such that $n = p\,q$.

```
Begin
  1.  x = 2⌈√n⌉
  2.  Choose a positive integer c s.t. gcd(c, n) = 1, say c=2
  3.  a = c^(−1) mod n
  4.  s = c^2 mod n
  5.  t = a^(n−x+1) mod n
  6.  If (t ==1) then
  7.      x = x/2
  8.      y = √(x^2 − n)
  9.      found=True
  10. Else
  11.     found=False
  12. End if
  13. While (found == False) do
  14.     t = t s
  15.     If (t > n) then
  16.         t = t mod n
  17.     End if
  18.     x = x + 2
  19.     If (t == 1) then
  20.         x = x/2
  21.         y = √(x^2 − n)
  22.         found=True
  23.     End if
  24. End while
  25. p = x + y
  26. q = x − y
End.
```

Remarks:

1) The original statements of Algorithm FnPS in lines 4 and 8 are if $(r < 0)$ and if $(r > 0)$, respectively, see [27]. To optimize the original algorithm, we replaced the two If-statements with two While-loops as in Algorithm FnPS. The performance of Algorithm FnPS with two inner While-loops is better than that with two If-statements, see Section IV.

2) Algorithm EF is slightly different than the algorithm in [20]. In [20], the Boolean condition *found* was not used. Instead, the statement at line 11 is *y* = 0.1, i.e., any initial value of *y* is considered to be a float number, and the statement at line 13 is "*y* is not an integer". The performance achieved by this modification is slightly better than that of the FE algorithm, but not significantly so.

IV. Experimental Evaluations

In this section, we discuss our experimental study of the nine different integer factorization algorithms that utilize Fermat's strategy. The study was conducted according to the methodology described in Section II and involved the evaluation of the performance of these algorithms over different numbers of bits and different differences between the two factors. The first subsection presents the specification of the hardware and software used in the implementation, and the

different values of the factors b and Δ that have an effect on the input data. The second subsection presents the results of applying the ideas mentioned in the remarks in Section III. The third subsection presents the measurement and analysis of the running times of the different integer factorization algorithms.

A. Experimental Specification

The experimental study of different factorization algorithms required the use of a high configuration of hardware. For this purpose, we used a Microsoft Azure cloud system. The system is able to run 32 threads in parallel with a processing speed of 2.6 GHz. The reasons for selecting this hardware platform are as follows: (1) The execution times of the factorization algorithms increase rapidly with increases in b and Δ; (2) a large number of instances are used to measure the running time for each algorithm for fixed values of b and Δ; and (3) experimentally, each algorithm is run sequentially, but we use different threads to execute different instances. However, in order to unload the system we used only a maximum of 16 threads.

As regards the implementation, we ran all the algorithms under Windows 2019 and used Microsoft Visual Studio 2019 to implement the algorithms using C++ language.

The specification of the data used in the evaluation of the different factorizing algorithms was as follows:

1) The values of b were 100, 200, 300, 400, and 500.

2) The value of Δ started with (b/4) and then we increased it incrementally by 5 bits until (b/4)+20. The reason that $\Delta = (b/4) + 20$ was set as the last value is that the running time for all the algorithms is very high and the time increases with an increase in b.

3) The value of t, i.e., the number of instances for the fixed values of b and Δ, was set as 100, except when $\Delta = (b/4) + 20$ where we considered t = 25, because the running time for each instance is greater than 1.5 hours for the best algorithm.

B. Experimental Comments

In this subsection, we illustrate the results that were achieved by applying the ideas mentioned in some of the remarks in Section III to optimize some of the statements in the previous algorithms.

However, first, the results in respect of the two versions of the FnPS algorithm, the If-statement and the While-loop, are presented and discussed. Table III shows the results of running two versions of the FnPS algorithm, on b = 100 and Δ =25, 30, 35, and 40. It is clear that the results indicate that the performance of the FnPS algorithm when using the While-loop is better than that when using the If-statement for all values of Δ. On average, the percentage of improvement in the performance of the FnPS algorithm with While-loop was 16.5% compared to its performance with the If-statement.

However, in general, the performance of the FnPS algorithm, even with While-loop, is very weak compared to that of the FF algorithm for two reasons. The first reason, from the theoretical point of view, is that the search space of the FnPS algorithm is very large compared to the search space of the FF algorithm. The second reason, from the practical perspective, is that the running time of the FF algorithm on $\Delta = 25$ and 30 is zero (see subsection IV.C), whereas the running time of the FnPS with the best case is 0.5 seconds. Additionally, the running time of the FF algorithm on $\Delta = \Delta_0 + 15 = 40$ is less than 1 minute, whereas the running time of the FnPS algorithm is greater than 5 hours. Finally, the results for the FnPS algorithm showed that increases in the value of b led to an increase in the search space, so we excluded this algorithm from the full comparison of all the integer factorization algorithms.

TABLE III. PERFORMANCE OF TWO VERSIONS OF FNPS WITH N = 100

Methods	Δ			
	25	**30**	**35**	**40**
FnPS with If-statement	0.6342 s	20.99 s	11.56 m	6.21 h
FnPS with While-loop	0.5146 s	17.57 s	9.73 m	5.25 h
Improvement %	18.86%	16.29%	15.83%	15.46%

Second, the results in respect of the FF1 algorithm are as follows. First, the results of running FF1 algorithm, on b = 100 and Δ =25 and 30, 35 are 0.001 and 0.002 seconds, respectively. Second, the results of running FF1 algorithm, on b = 100 and Δ =35 is greater than one hour in many cases without find the solution. As we see in the Subsection C, the running time for FF algorithm is faster than FF1 algorithm when $b \geq 100$, so we excluded this algorithm from the full comparison of all the integer factorization algorithms.

C. Results of the Comparison

Based on the results reported in Subsection IV.B, we compared seven of the nine algorithms, i.e., FF, FM10, FM20, FM1010, FM1020, FOE, and FE. The other two algorithms, FF1 and FnPS, were excluded from the final comparison because their performance was very poor.

The results of implementing the seven integer factorization algorithms using the data sets and platform described above are shown in Fig. 1, 2, 3, 4, and 5 for n = 100, 200, 300, 400, and 500, respectively. From the results shown in the figures, we can make several observations.

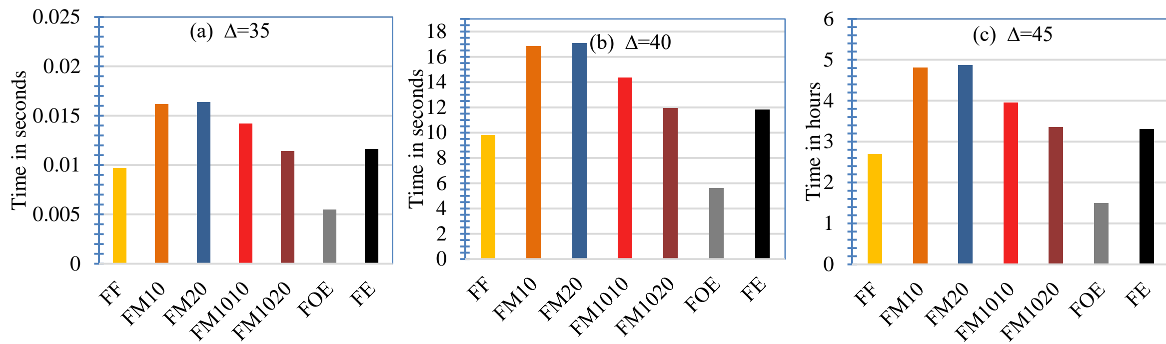

Fig. 1. Running Time for Fermat Algorithms when n=100.

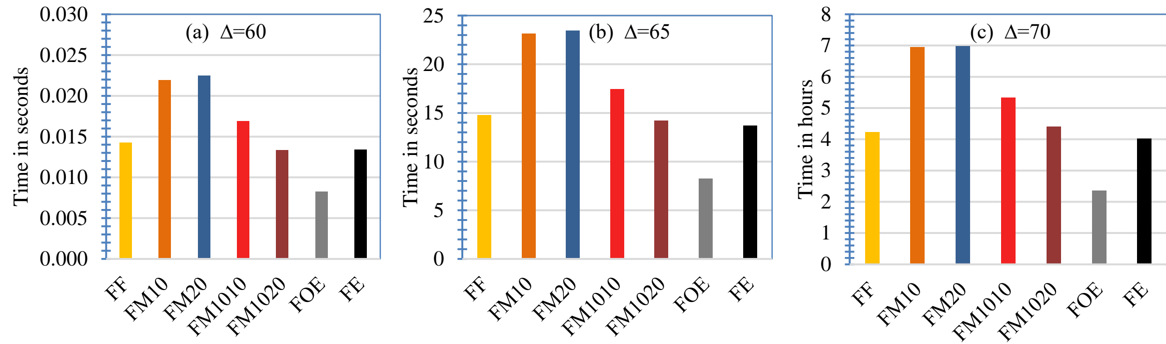

Fig. 2. Running Time for Fermat Algorithms when n=200.

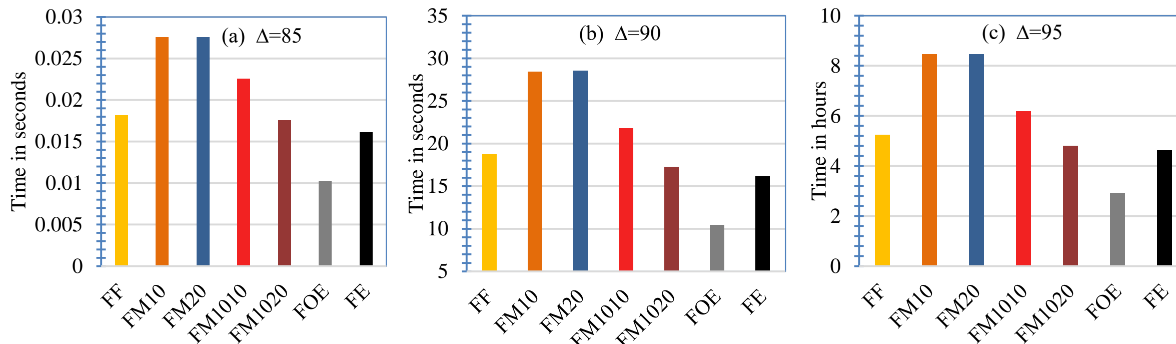

Fig. 3. Running Time for Fermat Algorithms when n=300.

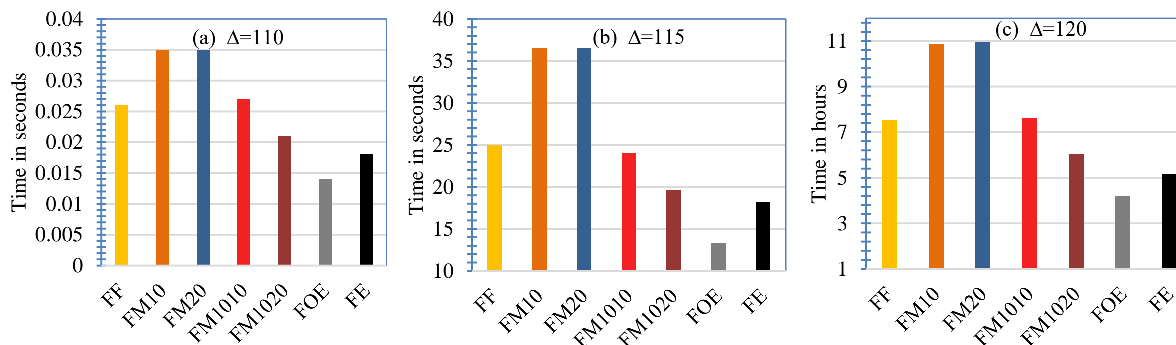

Fig. 4. Running Time for Fermat Algorithms when n=400.

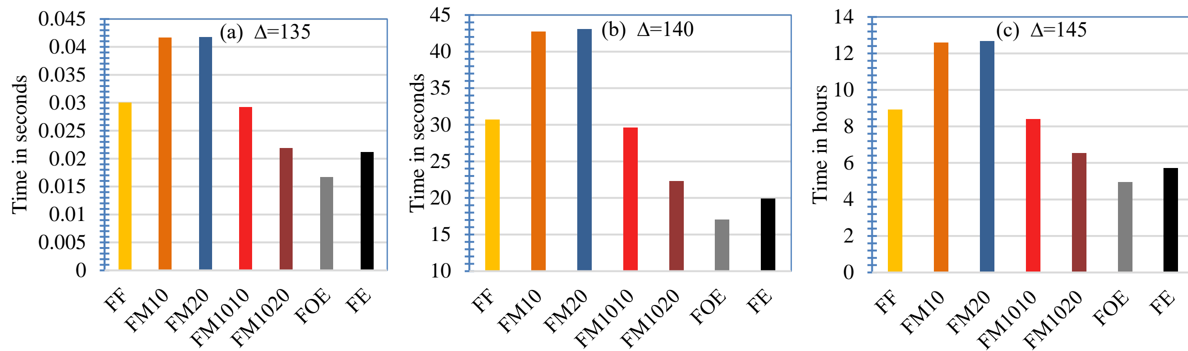

Fig. 5. Running Time for Fermat Algorithms when n=500.

First, the running time of all the algorithms is 0 seconds for all values of b and when $\Delta = b/4$, and $(b/4)+5$, except for the FE algorithm which has a very short runtime of 0.001 seconds when b = 400 and 500. For this reason, there are no data in the figures for $\Delta = b/4$, and $(b/4)+5$. This means that the running time for all seven of the Fermat-based algorithms is instantaneous in the case of $\Delta = b/4$, and $(b/4)+5$.

Second, for a fixed value of b with increasing values of Δ, the running time for all seven algorithms rapidly increases. We can estimate this factor by calculating the percentage of the increase in the running time between two consecutive Δ, Δ_1 and Δ_2, and a fixed value of b, where $\Delta_2 = \Delta_1 + 5$. By way of an example, Table IV shows this ratio for the FE algorithm when b = 300. The running times for the FE algorithm when b = 300 and Δ =85, 90, and 95 are 0.0161 seconds, 16.18 seconds, and 4.62 hours, respectively. The percentage increase in the running time when Δ_2=90 is approximately equal to 1000 times the running time when Δ_1 = 85, and the percentage of increase in the running time when Δ_2 = 95 is approximately equal to 1000 times the running time when Δ_1 = 90. This phenomenon is true for all values of b and for all seven of the integer factorization algorithms studied.

Third, the difference between the FM10 and FM20 algorithms is not significant and the two algorithms are near to equal for all values of b and Δ. On the other hand, the difference between the FM1010 and FM1020 algorithms is significant in respect of the average case. The FM1020 algorithm has better performance than the FM1010 algorithm by a percentage of 15.17%, 17.31%, 22.33%, 21.10%, and 22.17% for n = 100, 200, 300, 400, and 500, respectively. This means that the average percentage of improvement achieved by FM1020 is 19.62% when compared to FM1010.

Fourth, the performance of the FM10 and FM20 algorithms is weak compared to the other five algorithms for every b and Δ. The original Fermat algorithm, FF, has better performance than the FM10 and FM20 algorithms by almost 36% in respect of the average case.

Fifth, Table V shows the percentage of improvement achieved by each algorithm A compared to the FF algorithm for all values of b, where algorithm A is one of the following algorithms: FM10, FM20, FM1010, FM1020, FOE, and FE. Two types of value are presented in the table. The first value is a positive value, say ∝, which means that algorithm A has ∝ improvement compared to the FF algorithm, and the second value is a negative value, say -∝, which means that the FF algorithm has ∝ improvement compared to algorithm A.

Sixth, from Table V, it is clear that the percentage of dis-improvement of algorithm A decreases with an increase in the number of bits. For example, the FM1010 algorithm has a dis-improvement of 32.06%, 20.61%, 15.21%, and 1.04% for n = 100, 200, 300, and 400, respectively. Also, the percentage of improvement of algorithm A increases with an increase in the number of bits, except for the FOE algorithm. For example, the percentage of improvement achieved by the FE algorithm when compared to the FF algorithm is 4.89%, 11.83%, 31.78%, and 35.97% for n = 200, 300, 400, and 500, respectively.

Seventh, the FOE algorithm has two properties compared to the FF algorithm. The first is that the FOE algorithm has better performance compared to the FF algorithm for all values of b. The second is the percentage of improvement of the FOE algorithm compared to the FF algorithm is almost fixed and equal to 44% for all values of b. The reason for this property is that the number of iterations of the FOE algorithm is half that of the FF algorithm.

TABLE IV. RATIO OF RUNNING TIME BETWEEN TWO CONSECUTIVE Δ WHEN B = 300 FOR EACH ALGORITHM

Δ_2/Δ_1	**Fermat Algorithms**						
	FF	**FM10**	**FM20**	**FM1010**	**FM1020**	**FOE**	**FE**
90/85	1027	1029	1034	964	979	1016	1002
95/90	1009	1072	1068	1019	1001	1004	1031

TABLE V. PERCENTAGE OF IMPROVEMENT ACHIEVED BY EACH ALGORITHM COMPARED TO FF ALGORITHM

b	**Fermat Factorization Algorithms**					
	FM10	**FM20**	**FM1010**	**FM1020**	**FOE**	**FE**
100	- 44.12	- 44.80	- 32.06	- 19.91	43.99	- 18.70
200	- 39.12	- 39.42	- 20.61	- 3.99	44.38	4.89
300	- 38.06	- 38.13	- 15.21	8.40	44.27	11.83
400	- 30.42	- 30.92	- 1.04	20.27	44.24	31.78
500	- 29.27	- 29.61	5.87	26.73	44.38	35.97

Eighth, Fig. 6 displays the behavior of all the algorithms for the average case for all different values of b. The same behavior occurs if we fix the value of Δ and change the value of b. It is clear that the best integer factorization algorithm based on Fermat's strategy for all data sets is the algorithm that is based on sieving odd and even numbers, i.e., the FOE algorithm. The second and third best algorithms are FE and FM1020, respectively. Note that the two curves for the FM10 and FM20 algorithms coincide.

Ninth, Fig. 7 displays the running times for $t = 100$ instances for each algorithm A when $b = 500$ and $\Delta = 140$, where algorithm A is one of the following algorithms: FM1010, FM1020, FOE, and FE. From the figure, we can observe the following: (1) The running time of each algorithm A is varied and based on the value of n, except for the FE algorithm whose behavior is almost fixed; and (2) the running time of the FM1010 algorithm is slightly faster than that of the FF algorithm on average, as in Fig. 5(b), but in many instances the running time of the FM1010 algorithm is longer than the FF algorithm.

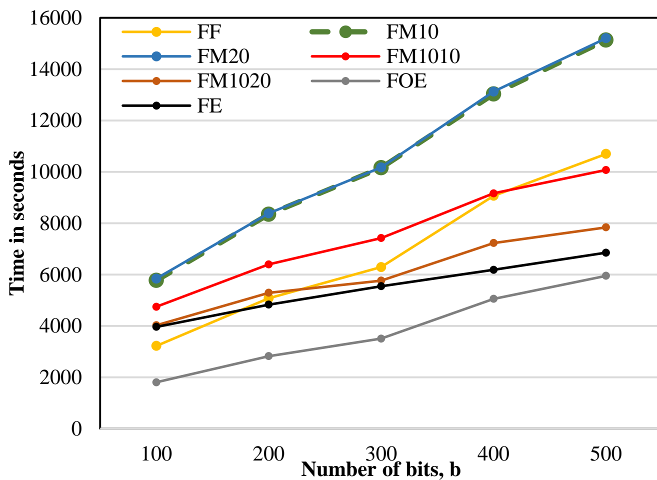

Fig. 6. Average Running Time of Fermat algorithms over Different n.

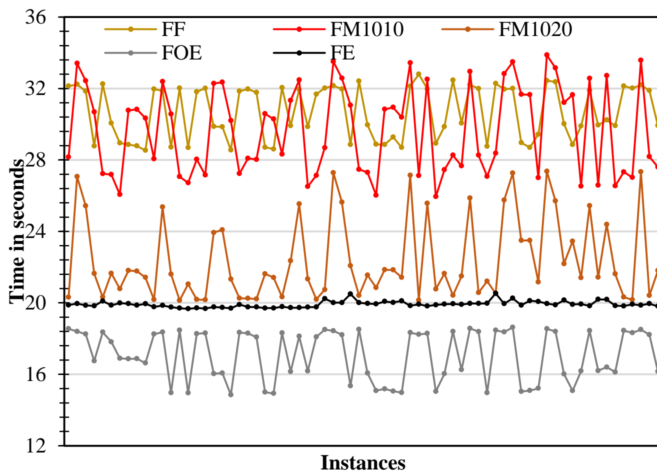

Fig. 7. Behavior of FF, FM1010, FM1020, FOE and FE algorithms when $b = 500$ and $\Delta = 140$.

V. Conclusion and Open Problems

In this work, we addressed the integer factorization problem that involves finding the two prime factors for an odd composite number. The inherent challenge of this problem is that no polynomial time has yet been found. The problem is used in cryptography to break the RSA cryptosystem. We classified the different Fermat factorization algorithms into three groups according to the integer factorization methods that were used. The algorithms were studied experimentally to measure their performance according to two factors: (1) the size of the odd composite factor, b bits, and (2) the difference between two factors, Δ bits. The experimental study was conducted on data sets consisting of b =100, 200, 300, 400, and 500, and $\Delta = b/4$, $(b/4) + 5$, $(b/4) + 10$, $(b/4) + 15$, and $(b/4) + 20$. The results of the experimental study showed that the algorithm based on sieving using the odd and even property performed the fastest factorization when applied to data sets of different sizes of odd composite numbers and different differences between the two factors with a percentage improvement of 44% compared to the original Fermat factorization algorithm. Also, the algorithm based on Euler's theorem exhibited a good level of performance compared with the Fermat factorization algorithm.

There are open questions remain related to this study. (1) What is the behavior of FM1020, FOE and FE algorithms when $b > 500$ bits? (2) How to use high-performance computing such as graphics processing unit (GPU) to speedup the running time for the best modified Fermat algorithm. (3) What is the effect of using n mod m on FM1020 algorithm, when m>20.

Acknowledgment

This work has been funded by Scientific Research Deanship at the University of Ha'il – Saudi Arabia through project number RG-191309.

References

[1] K. Balasubramanian, and M. Rajakani, "Algorithmic strategies for solving complex problems in cryptography," IGI global, 2017.

[2] S. Yan, "Primality testing and integer factorization in public-key cryptography," Springer, 2009.

[3] M. Daniel, "A survey of fast exponentiation methods," J. Algorithm vol. 27, no. 1, pp. 129–146, 1998.

[4] K. Fathy, H. Bahig, and A. Ragab, "A fast parallel modular exponentiation algorithm," Arabian Journal for Science and Engineering, vol. 43, pp. 903–911, 2018.

[5] R. Crandall, and C. Pomerance, "Prime numbers: a computational perspective," 2nd Ed., Springer, 2005.

[6] O. Akchiche, and O Khadir, "Factoring RSA moduli with primes sharing bits in the middle," Applicable Algebra in Engineering, Communication and Computing, vol. 29, pp. 245–259, 2018.

[7] H. Bahig, D. Nassr, and A. Bhery, "Factoring RSA modulus with primes not necessarily sharing least significant bits," Applied Mathematics and Information Sciences, vol. 11, no. 1, pp. 243-249, 2017.

[8] H. Bahig, D. Nassr, A.Bhery and A. Nitaj, "A unified method for private exponent attacks on RSA using lattices," International Journal of Foundations of Computer Science, vol. 31, no. 2, pp. 207-231, 2020.

[9] R. Steinfeld, and Y. Zheng, "On the security of RSA with primes sharing least-significant bits," Applicable Algebra in Engineering, Communication and Computing, vol. 15, pp. 179–200, 2004.

[10] A. Lenstra, "Integer factoring," Designs, Codes and Cryptography, vol. 19, pp. 101–128, 2000.

[11] J Jormakka, "On finding Fermat's pairs," Journal of Discrete Mathematical Sciences & Cryptography, vol. 10, no. 3, pp. 401-413, 2006.

[12] B. de Weger, "Cryptanalysis of RSA with small prime difference," Applicable Algebra in Engineering, Communication and Computing, vol. 13, no. 1, pp. 17–28, 2002.

[13] H. Bahig, H. Bahig, and Y Kotb, "Fermat factorization using a multi-core system," International Journal of Advanced Computer Science and Applications, vol. 11, no. 4, pp. 323-330, 2020.

[14] R. Erra, and C. Grenier, "The Fermat factorization method revisited," IACR Cryptology ePrint Archive, 318 2009.

[15] W. Hart, "A one line factorization algorithm," Journal of the Australian Mathematical Society, vol. 94, pp. 61-69, 2012.

[16] M. Hittmeir, "Deterministic factorization of sums and differences of powers," Mathematics of Computation, vol. 86, no. 308, pp. 2947–2954, 2017.

[17] J. Mckee, "Speeding Fermat's factorization method," Mathematics of Computation, vol. 68, no. 228, pp. 1729–1737, 1999.

[18] B. Randall, "Fingers find Fermat's factorization most probable," The Mathematical Gazette, vol. 99, no. 544, pp. 452-458, 2014.

[19] P. Shiu, "Fermat's method of factorization," The Mathematical Gazette, vol. 99, no. 544, pp. 97-103, 2015.

[20] K. Somsuk, "The new integer factorization algorithm based on Fermat's factorization algorithm and euler's theorem," International Journal of Electrical and Computer Engineering, vol. 10, no. 2, pp. 1469-1476, 2020.

[21] K. Somsuk, and S. Kasemvilas, "MVFactor: A method to decrease processing time for factorization algorithm," Proceedings of 17th International Computer Science and Engineering Conference, Thailand, 2013, pp. 339-342.

[22] K. Somsuk and S. Kasemvilas, "MFFV2 and MNQSV2: improved factorization algorithms," Proceeding of 4th International Conference on Information Science and Applications, 2013, pp. 327 – 329.

[23] K. Somsuk and S. Kasemvilas, "Possible prime modified Fermat factorization: new improved integer factorization to decrease computation time for breaking RSA," Proceedings of the 10th International Conference on Computing and Information Technology. Advances in Intelligent Systems and Computing, vol. 265, pp. 325-334, 2014.

[24] K. Somsuk and S. Kasemvilas, "MFFV3: An Improved Integer Factorization Algorithm to Increase Computation Speed", 5th International Engineering Conference 2014, pp. 1432 – 1436, 2014.

[25] K.Somsuk, "A new modified integer factorization algorithm using integer mod 20's technique," Proceedings of the 18 International Computer Science and Engineering Conference, Thailand, 2014, pp. 312-316.

[26] K. Somsuk and K. Tientanopajai, "Improving Fermat factorization algorithm by dividing modulus into three forms," KKU Engineering Journal, vol. 43, no. S2, pp. 350-353, 2016.

[27] K. Somsuk, and K. Tientanopajai, "An improvement of Fermat's factorization by considering the last m digits of modulus to decrease computation time," International Journal of Network Security, vol. 19, pp. 99-111, 2017.

[28] M. Wu, R.Tso, and H. Sun, "On the improvement of Fermat factorization using a continued fraction technique," Future Generation Computer Systems, vol. 30, no. 1, pp. 162-168, 2014.

[29] G.Xiang, "Fermat's method of factorization," Applied Probability Trust, vol. 36, no. 2, pp. 34-35, 2004.

[30] R. Sakellariou, "Parallel algorithms for integer factorization," Advances on Computer Mathematics and Its Applications, pp. 288-295, 1993.

[31] G. Hiary, "A deterministic algorithm for integer factorization," Mathematics of Computation, vol. 85, pp. 2065-2069, 2016.

[32] E. Costa, and D. Harvey, "Faster deterministic integer factorization," Mathematics of Computation, vol. 83, pp. 339-345, 2014.

[33] R. Sakellariou, "Parallel algorithms for integer factorization," Advances on Computer Mathematics and Its Applications, pp. 288-295, 1993.

[34] GMP library, The GNU multiple precision arithmetic library. https:// gmplib.org/, 2020.

[35] H. M. Bahig, A. Alghadhban, M. A. Mahdi, K. A. Alutaibi, H. M. Bahig, "Speeding up the multiplication algorithm for large integers", Engineering, Technology & Applied Science Research, vol 10, no. 6, pp 6533-6541, 2020.

[36] H. Bahig, H. Bahig, and K. Fathy, "Fast and scalable algorithm for product large data on multicore system," Concurrency and Computation: Practice and Experience, https://doi.org/10.1002/cpe.5259, online published 2019.